# Chapter 6

# ASM

Egon BÖRGER, Angelo GARGANTINI and Elvinia RICCOBENE

## 6.1 Overview of the ASM

The *Abstract State Machine* (ASM) method is a systems engineering method that guides the development of software and embedded hardware-software systems seamlessly from requirements capture to their implementation. Within a single precise yet simple conceptual framework, the ASM method supports and uniformly integrates the major software life cycle activities of the development of complex software systems. The process of *requirements capture* results into constructing rigorous *ground models* which are precise but concise high-level system blueprints ("system contracts"), formulated in domain-specific terms, using an application-oriented language which can be understood by all stakeholders. From the ground model, by piecemeal, systematically documented detailing of abstract models via stepwise refined models to code, the *architectural and component design* is obtained in a way which bridges the gap between specification and code. The resulting *documentation* maps the structure of the blueprint to compilable code, providing explicit descriptions of the software structure and of the major design decisions, besides a road map for system *(re-)use* and *maintenance*.

On the basis of a systematic separation of different concerns (e.g. design from analysis, orthogonal design decisions, multiple levels of definitional or proof detail, etc.), the ASM method allows a nowadays widely-requested *modeling technique* which integrates dynamic (*operational*) and static (*declarative*) descriptions, and an *analysis technique* that combines *validation* (by simulation and testing) and *verification* methods at any desired level of detail.

Even if the ASM method comes with a rigorous scientific foundation [BÖR 03], the practitioner needs no special training to use the ASM method since Abstract State Machines are a simple extension of Finite State Machines, obtained by replacing

unstructured "internal" control states by states comprising arbitrarily complex data [BÖR 05], and can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures. Control state ASMs, a basic class of Abstract State Machines, inherit from FSMs their standard graphical notation (see [BÖR 03, Figure 7.1]). Similarly, UML activity diagrams pass to their ASM models their graphical notation (see [BÖR 03, Figure 6.18, 6.19]), as do SDL programs or Petri nets to their ASM models.

A complete introduction on the ASM method can be found in [BÖR 03], together with a presentation of the great variety of its successful applications in different fields as: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemes and compiler back-ends, etc.

## 6.2 Requirements capture and specification of case 1

We formulate seven categories of questions to be used as guidelines for the specification task leading from loosely formulated requirements to accurate, application-domain-oriented ground models. The questions are prompted by the application of the ASM method to the case 1 of the invoicing order system, although similar questions should be posed when using the ASM method for requirements capture and specification of other systems. Answers are preceded by explanations of some relevant ASM concepts.

### 6.2.1 Identifying the agents

An ASM can be intuitively viewed as pseudo-code or Virtual Machine program working on abstract data. The notion of ASMs moved from a definition which formalises simultaneous parallel actions of a single agent, either in an atomic way (*Basic ASMs*) or in a structured and recursive way (*Turbo ASMs*), to a generalisation where multiple agents interact in a synchronous/asynchronous manner[1] (*Synchronous/Asynchronous Multi-Agent ASMs*). The context in which an agent machine computes is represented by an external agent called *environment*.

**Question 1:** Who are the system *agents* and what are their relations? In particular, what is the relation between the system and its environment?

**Answer:** *R1* says that "the subject is to invoice orders". This leads us to define the *invoicing orders* specification in terms of a single-agent machine which may dispose of potentially unrestricted non-determinism and parallelism (appearing in the form of the "choose" and "forall" rules defined below) with flat programs (Basic ASM) or structured versions (Turbo ASM).

[1] For details and references on the treatment of *concurrency* in the ASM framework and on concurrent ASMs modeling threads in Java/C#, Petri nets, SDL, UML activity diagrams and state machines, etc., see [BÖR 03, Chapter 6].

### 6.2.2 Identifying the states

An ASM *state* models a machine state, i.e. the collection of elements and objects the machine "knows", and the functions and predicates it uses to manipulate them. Mathematically, a *state* is defined as an algebraic structure, where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions*) and *predicates* (attributes or relations).

For the evaluation of terms and formulas in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used. Without loss of generality we usually treat predicates as characteristic functions and constants as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x)$ = *undef* with a fixed special value *undef* as $f(x)$ being undefined. The reader who is not familiar with this notion of structure may view a state as a "database of functions" (namely a set of function tables).

**Question 2:** What are the system states? What are the domains of objects and what are the functions, predicates and relations defined on them? This question is stressed by the object-oriented approach to system design[2].

**Answer:** By *R1* there is a set Orders and by *R2* there is a function orderState which yields the state of each order, which can be *invoiced* or *pending*. By *R3* there are two functions, referencedProduct[3] representing the product referenced in an order and orderQuantity, which returns the quantity in the order and which, by *R4*, is not injective, not constant. By *R3* we need a set Quantity (subset of Natural) to denote the quantity values, while by *R5* there is a function stockQuantity which represents the quantity of products in stock.

### 6.2.3 Identifying static and dynamic parts of the states

In support of the principles of separation of concerns, information hiding, data abstraction, modularisation and stepwise refinement, the ASM method makes a systematic distinction between *basic* functions which are taken for granted (typically those forming the basic signature of an ASM) and *derived* functions (auxiliary functions coming with a specification or computation mechanism given in terms of basic functions), together with a classification of basic functions into *static* and *dynamic* ones and of the dynamic ones into *monitored* (only read), *controlled* (read and write), *shared* and *output* (only write) functions. This functions classification reflects the different roles these functions can assume in a given machine. *Static* functions never change during

[2] For details on object-oriented ASMs, their theory (developed mainly in the work by Zamulin), their use for modeling object-oriented databases and languages, e.g. C++, Java, C#, SDL, and their incorporation into the language AsmL of .NET-executable ASMs, see [BÖR 03, Chapter 9].

[3] To allow an order to reference to several products, we should introduce a single function referencedProductQuantity: Orders $\times$ Products -> Quantity, which yields the quantity of products in an order (undef in case a product is not referenced in a given order).

any run of the machine so that their values for given arguments do not depend on the states of the machine; *dynamic* functions may change as a consequence of agent actions (or *updates*, see definition below) or by the *environment*, so that their values may depend on the states of the machine. By definition static functions can be thought of as given by the initial state, so that, where appropriate, handling them can be clearly separated from the description of the system dynamics. Whether the meaning of these functions is determined by a mere signature description, by axiomatic constraints, by an abstract specification or by an explicit or recursive definition depends on the degree of information-hiding the specifier wants to realize. Static $0$-ary functions represent *constants*, whereas with dynamic $0$-ary functions one can model *variables* of programming (not to be confused with logical variables). *Controlled* functions are dynamic functions which are directly updatable by and only by the machine instructions (known as *transition rules*: see below). Therefore, these functions are the ones which constitute the internally controlled part of the dynamic state of the machine; they are not updatable by the environment (or more generally by another agent in the case of a multi-agent machine). *Monitored* functions are dynamic functions which are read but not updated by a machine and directly updatable only by the environment (or more generally by other agents). These monitored functions constitute the externally controlled part of a machine state. As with static functions, the specification of monitored functions is open to any appropriate method. The only (but crucial) assumption made is that in a given state the values of all monitored functions are determined. Combinations of internal and external control are captured by interaction or *shared* functions that can be read and are directly updatable by more than one machine (so that typically a protocol is needed to guarantee consistency of updates). *Output* functions are updated but not read by a machine and are typically monitored by other machines or by the environment.

**Question 3:** What are the static and the dynamic parts of states? Who can update the dynamic functions?

**Answer:** By *R6a* the set Orders is static. By *R2* and *R5* the function orderState is dynamic and controlled by the system. By *R3* and *R6a* referencedProduct and orderQuantity are both static. By *R6a* the function stockQuantity is dynamic – a static interpretation is not reasonable – but it is unclear if the function is updated by the environment or by the system or by both of them (shared function). We make the *assumption* that *the stock is only updated by the system when it invoices an order*. The set of products and of quantities are assumed to be static. For writing down ASMs we use the AsmM language [ASMM] which has been derived from a metamodel of the ASMs and is endowed with a BNF grammar [SCA 05] and a syntax checker.

```
asm orderSystemCase1
 signature:
    static abstract domain Orders
    enum domain OrderStatus = { INVOICED | PENDING }
```

**static abstract domain** Products
**static domain** Quantity **subsetof** Natural
**static** referencedProduct: Orders -> Products
**dynamic controlled** orderState: Orders -> OrderStatus
**static** orderQuantity: Orders -> Quantity
**dynamic controlled** stockQuantity: Product -> Quantity

### 6.2.4 Identifying the transitions

Basic ASMs are finite sets of so-called *transition rules* of the form:

**if** *Condition* **then** *Updates*

which model the actions performed by the machine to manipulate elements of its domains and which result in a new state. The *Condition* (also called *guard*) under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to true or false. *Updates* is a finite set of assignments of the form $f(t_1, t_2, \ldots, t_n) := t$, whose execution is to be understood as changing (or defining, if there was none) in parallel the value of the occurring functions $f$ at the indicated arguments to the indicated value. More precisely, in the given state, first all parameters $t_i$, $t$ are evaluated to their values, say $v_i$, $v$, then the value of $f(v_1, v_2, \ldots, v_n)$ is updated to $v$, which represents the value of $f(v_1, v_2, \ldots, v_n)$ in the next state. Such pairs of a function name $f$, which is fixed by the signature, and an optional argument $(v_1, v_2, \ldots, v_n)$, which is formed by a list of dynamic parameter values $v_i$ of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms. Location-value pairs (*loc, v*) are called *updates* and represent the basic units of state change.

*Non-determinism* is a convenient way to abstract from details of scheduling of rule executions. It can be expressed by rules of the form:

**choose** *v* **in** *D* **with** $G_v$ **do** $R(v)$

where *v* is a variable, *D* is a domain in which *v* takes its value, $G_v$ is a term representing a boolean condition over $v$, and $R(v)$ is a transition rule which contains the free variable $v$. The meaning of such an ASM rule is to execute rule $R(v)$ with an arbitrary $v$ chosen in $D$ among those satisfying the selection property $G_v$. If there exists no such $v$, nothing is done.

**Question 4:** How and by which transitions (actions) do system states evolve? Under which conditions (guards) do the state transitions (actions) of single agents happen and what is their effect on the state? What is supposed to happen if those conditions are not satisfied?

**Answer:** By *R2* and *R5* there is only one transition to change the state of an order. It remains open whether the invoicing is done only for one order at a time, simultaneously for all orders, or only for a subset of orders (with a synchronisation for concurrent access of the same product by different orders). In case

the update is meant to be made for one order at a time, it remains unspecified in which succession and with what successful termination or abruption mechanism this should be realized. The time model (duration of invoicing) is also not mentioned.
Modulo all those missing pieces of information, one can nevertheless reason upon possible rules for invoicing orders. A single-order rule can be formalised as follows[4]. Per step at most one order is invoiced, with an unspecified schedule (thus also not taking into account any arrival time of orders) and with an abstract deletion function:

```
rule r_InvoiceSingleOrder =
  choose $o in Orders with orderState($o) = PENDING and
     orderQuantity($o) <= stockQuantity(referencedProduct($o))
     do par
        orderState($o) := INVOICED
          r_DeleteStock[referencedProduct($o),orderQuantity($o)]
     endpar
```

Under the assumptions that stockQuantity is updated only by invoicing and only one order is processed at a time, the deletion function can be refined by the following macro rule:

```
rule macro r_DeleteStock($p in Products, $q in Quantity) =
        stockQuantity($p):= stockQuantity($p) - $q
```

The rule InvoiceSingleOrder has the disadvantage to invoice an order at a time, while some strategies could admit that the system can simultaneously invoice a certain number of orders at a time, if any. Simultaneous execution provides a convenient way to abstract from sequentiality where it is irrelevant for an intended design. In the ASM execution model, this *synchronous parallelism* is enhanced by the following notation to express the simultaneous execution of a rule $R$ for each $v$ satisfying a given condition $G$ (where typically $v$ will have some free occurrences in $R$ which are bound by the quantifier):

**forall** $v$ **in** $D$ **with** $G_v$ **do** $R(v)$

**Question 5:** Could the system actions be parallelised anyhow? Namely, in the case of invoicing orders, can the system invoice several orders in one step?

**Answer:** To speed up invoicing of orders, parallelism can be exploited in two directions. A first strategy consists of selecting a given product (possibly in a non-deterministic way) and then simultaneously invoicing all the corresponding orders, if possible. An alternative policy could be selecting, still non-deterministically, a set of orders to be invoiced in parallel.

[4] In AsmM a rule identifier begins with r_ and a logical variable identifier starts with $.

In case all orders for one product are simultaneously invoiced (or none if the stock cannot satisfy the request), an "all-or-none" strategy can be expressed by the following rule InvoiceAllOrNone which makes use of a function pendingOrders yielding the set of pending orders for a certain product, and of a (static) function totalQuantity returning the total quantity of a set of orders. The functions are defined below the rule:

```
rule r_InvoiceAllOrNone =
  choose $product in Products do
    let $pending = pendingOrders($product),
        $total = totalQuantity($pending) in
    if $total <= stockQuantity($product) then par
       forall $ord in $pending do orderState($ord) := INVOICED
     r_DeleteStock[$product, $total]
    endpar endif
```

where:

```
static function pendingOrders($p in Products): Powerset(Orders) =
   {$o | $o in Orders with orderState($o) = PENDING and
                          referencedProduct($o) = $p}

static function totalQuantity($so in Powerset(Orders)): Quantity =
   if (isEmpty($so)) then 0
   else let $first = first(asSequence($so) in
    quantity($first) + totalQuantity(excluding($so,$first))
   endif
```

The previous definition of DeleteStock can be kept in this case as well. Indeed, the cumulative effect of updating the product quantity in stock is obtained by using the total quantity of the set of invoiced orders.
To avoid the system deadlock when the stock cannot satisfy any request, we formalise, by the following rule InvoiceOrdersForOneProduct, the second strategy introducing some non-determinism in the choice of a set of pending orders which can be invoiced according to the available quantity in stock:

```
rule r_InvoiceOrdersForOneProduct =
 choose $product in Products do
  let $pending = pendingOrders($product) in
  choose $ordSet in Powerset($pending)
     with totalQuantity($ordSet) <= stockQuantity($product)
   do par
     forall $ord in $ordSet do orderState($ord) := INVOICED
     r_DeleteStock[$product, totalQuantity($ordSet)]
   endpar
```

To parallelise invoicing orders over all products, a slight variant of the previous rule can be obtained replacing choose $product in Products with forall $product in Products. To further maximise a product quantity invoiced at the time, a new strategy is formalised by the rule InvoiceMaxOrdersForOneProduct. It consists of choosing a maximal invoicable subset of simultaneously invoiced pending orders for the same product. For this rule we need to define a static function maxQuantitySubsets defined on Powerset(Powerset(Orders)) to Powerset(Powerset(Orders)) which, given a set of set of orders, returns the set of all the sets having a maximum quantity:

```
rule r_InvoiceMaxOrdersForOneProduct =
 choose $product in Products do
   let $pending = pendingOrders($product),
       $invoicable = {$o | $o in Powerset($pending)
         with totalQuantity($o) <= stockQuantity($product) } in
  choose $ordSet in maxQuantitySubsets($invoicable) do par
    forall $ord in $ordSet do orderState($ord) := INVOICED
    r_DeleteStock[$product, totalQuantity($ordSet)]
 endpar
```

If the user requests a selection strategy which is not driven by a first choice of a product, another possible policy is to choose a set of pending orders, with enough referenced products in the stock, to be simultaneously invoiced. We reckon that this policy matches the intended behavior of the system better than the previous policies. The rule InvoiceOrders uses a predicate invoicable which is true on a set of pending orders with enough quantity of requested products in the stack, and a function refProducts which yields the set of all products referenced in a set of orders (the function is recursively defined below):

```
rule r_InvoiceOrders =
  choose $oSet in Powerset(Orders) with invoicable($oSet)
   do par
    forall $ord in $oSet do orderState($ord) := INVOICED
    forall $p in refProducts($oSet) do
              r_DeleteStock[$p, totalQuantity($oSet,$p)]
   endpar

static function invoicable($so in Powerset(Orders)) : Boolean =
  forall $o in $so with orderState($o) = PENDING and
   forall $p in Products with totalQuantity($so,$p) <= stockQuan-
    tity($p)

static function
      refProducts($so in Powerset(Orders)) : Powerset(Products) =
   if (isEmpty($so)) then {}
   else let $first = first(asSequence($so) in
           including(refProducts(excluding($so,$first)),
                              referencedProduct($first))
   endif
```

Note that in all the previous examples, the non-deterministic selection of the orders to invoice could be performed by a monitored function which would formalise the user selection of a set of orders or the results of a particular scheduling algorithm.

### 6.2.5 Identifying the initial and final states

The *computation* of an ASM is defined in the standard way transition system *runs* are defined. Applying one step of the abstract machine M to a state $S$ produces as next state another state $S'$ of the same signature, which is obtained as follows: first evaluate in $S$, using the standard interpretation of classical logic, all the guards of all the rules of $M$, then compute in $S$, for each of the rules of $M$ whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule; finally replace, simultaneously for each rule and for all the locations in question, the previous $S$-function value by the newly computed value if no two required updates contradict each other. The state $S'$ thus obtained differs from $S$ by the new values for those functions at those arguments where the values are updated by a rule of $M$ which could fire in $S$. The effect of an ASM $M$, started in an arbitrary *initial* state $S$ (generally provided by the user), is to repeatedly apply one step of $M$ as long as an $M$-rule can fire. Such a machine terminates (in a *final* state) only if no rule is applicable anymore (and if the monitored functions do not change in the state where the guards of all the $M$-rules are false).

**Question 6:** What is the initialisation of the system and who provides it? Are there termination conditions and, if so, how are they determined? What is the relation between initialisation/termination and input/output?

**Answer:** No explicit initialisation is specified, although one can assume that all the orders are initially pending:

```
default init s_1: function orderState( $o in Orders) = PENDING
```

No termination condition is given either. We assume that the system keeps to invoice orders as long as there are orders which can be invoiced (i.e. they are pending and there is enough product quantity in stock).

### 6.2.6 Exceptions handling and robustness

Usually, an ASM specification captures requirements concerning error handling by transition rules guarded by events[5] occurring in erroneous situations, and therefore separated by transition rules describing the normal machine execution.

[5] For details on event-driven ASMs, see [BÖR 03, section 6.5], which includes UML activity diagram ASMs. Event-driven ASMs also comprise Petri net ASMs [BÖR 03, sections 6.1,7.1.2], Abstract State Processes and Event-B ASMs [BÖR 03, section 4.2].

Furthermore, Turbo ASMs (see page 116) support exception-handling techniques to treat errors due to inconsistent updates. In Turbo ASMs, an abstract method for catching an inconsistent update set and of executing error handling rules is given by the **try-catch** rule. Let $T$ be a set of terms. The semantics of **try** $P$ **catch** $T$ $Q$ is to execute $P$, if the update set of $P$ is consistent on the locations determined by elements of $T$, otherwise $Q$ is executed.

**Question 7:** Which forms of erroneous use are to be foreseen and which exception handling mechanisms should be installed to catch them? What are the desired robustness features?

**Answer:** Since no exceptional computations are mentioned in the requirements and no inconsistent updates are allowed by the specification (see Question 8), we do not make use of the techniques supported by the ASM method to the error-handling purpose.

### 6.2.7 Identifying the desired properties (validation/verification)

The notion of ASM run makes the mechanical execution of ASM models possible, and various tools have been built for model *validation* by simulation and testing (see section 8.3 of [BÖR 03]). Furthermore, the rigorous mathematical definition of ASMs allows any standard mathematical *verification* technique to prove ASM model properties: from proof sketches over traditional or formalized mathematical proofs to tool supported interactive or automatic theorem proving or model checking (see sections 8.1 and 8.2 of [BÖR 03]).

**Question 8:** Is the system description complete and consistent?

**Answer:** *Completeness* with respect to the requirements can be verified for example by checking that every requirement has been analysed and captured by our specification. To validate a specification and its completeness with respect to user needs, it is important that the specification can be simulated by the user to uncover missing bits and pieces in the ground model. An ASM is *consistent* if it always performs consistent updates (i.e. it never tries to update in the same step the same location with different values). In our case there is a single rule which invoices one or more orders by updating simultaneously the status of the orders and the stock quantity. Since this single rule updates independently the status of different orders and updates the stock quantity of different products by means of a total quantity function which computes the cumulative effect of invoiced orders on the stock, the updates are always consistent.

**Question 9:** What are the system assumptions and what are the desired system properties? What do the requirements say about the state of the system?

**Answer:** No explicit assumptions or desired properties are given in the original specification. Through the requirements capture we have introduced several assumptions to fill missing information. For example, we have assumed that stockQuantity is updated only by the rule which invoices orders. Other assumptions

can be introduced by means of auxiliary axioms. For example, the assumption that the quantity in every order must be greater than 0 is formalised as:

**axiom over** orderQuantity:
forall $o in Orders with orderQuantity($o) > 0

We have stated the following desired properties which express state invariants and correctness conditions. The first one states that the stock quantity is always greater than 0, i.e. the system cannot over invoice orders:

**axiom over** stockQuantity:
forall $p in Products with stockQuantity($p)>=0

Another property is that the state of every order is either pending or invoiced, but never undefined:

**axiom over** orderState:
forall $o in Orders with orderState($o) != undef

These properties have been proved by the method proposed in [GAR 00] and based on the theorem prover PVS. We report here only a sketch of the resulting encoding in PVS of the ASM for the order system. The controlled part of an ASM state is encoded in PVS as a record of functions representing the controlled ASM functions:

```
CTRLSTATE: TYPE =[#orderState:[Orders -> OrderStatus],
         stockQuantity :  [Products -> Quantity] #]
```

Each rule is a function that given a current state `c` and an intermediate controlled state `ctrl` returns a new controlled state in which the updates have been applied. The rule InvoiceSingleOrder is translated as follows, where the choose construct is substituted by the dynamic function `choose_order` (as explained in [GAR 00]):

```
InvoiceSingleOrder(c,ctrl) :  CTRLSTATE =
 let ord = choose_order(c) in
 let prod = referencedProduct(ord) in
 if orderState(c)(ord) = PENDING then ctrl with [
  orderState := orderState(c) with [(ord):= INVOICED],
  stockQuantity := stockQuantity(c) with [(prod) :=
       stockQuantity(c)(prod) + orderQuantity(ord)]]
 else ctrl endif
```

The properties are encoded as functions from `STATE` to `bool`. For example, the second property above is encoded as:

```
prop2(s:  STATE) :bool =
     forall (o:Orders):  orderState(s) (o) /= undef
```

and it is proved using induction and very simple PVS strategies.
Other more complex properties, which are not state invariants but which refer to execution paths, cannot be encoded in our verification method yet. For these properties, temporal logic and model checkers [DEL 00] could be used, although assumptions about the finiteness of the domains are necessary and uninterpreted domains are not allowed. For example, one may want to express that *an order o is eventually invoiced if it refers to a product available in the stock in enough quantity*. In CTL, this can be expressed as:

```
AF( AG( orderState(o) = INVOICED or
orderQuantity(o) > stockQuantity(referencedProduct(o)))
```

### 6.3 Requirements capture and specification of case 2

In this section we formulate for the answers to the very questions of case 1 only the changes needed for case 2.

**Question 10:** Who are the system *agents*?

**Answer:** The informal description does not specify the agents for dynamic manipulation of orders, stock and products, how they interact for shared data (namely the elements of Orders and the function stockQuantity), whether they act independently or following a schedule. For the sake of simplicity we assume that our system still has only one agent which performs all the requested actions. The main program executed by the agent (i.e. its main rule) will take care of the synchronisation of actions to avoid inconsistencies.

**Question 11:** What are the system *states*? What are the domains of objects and what are the functions, predicates and relations defined on them?

**Answer:** The domains Orders and Products and all the functions for case 1 remain. For the new operations of this case, we introduce the following three monitored functions that respectively yield the sequence of orders to add (as a sequence of pairs product and quantity), the sequence of orders to cancel, and the new quantities to add in the stock (as sequence of pairs product and quantity again):

**monitored** newOrders: Seq(Prod(Products,Quantity))
**monitored** ordersToCancel: Seq(Orders)
**monitored** newItems: Seq(Prod(Products,Quantity))

The value of these functions may be determined by the user or be the output produced by other system components in charge of computing orders to add or cancel and items to entry in the stock. They are considered system inputs.
The requirements do not specify whether a canceled order must be completely deleted from the system or whether it must be kept and marked as canceled. We assume that canceled orders are not deleted and their status changed to CANCELED. Therefore, the order status is modified as:

**enum domain** OrderStatus = {INVOICED | PENDING |CANCELED}

**Question 12:** What is the *classification* of domains and functions?

**Answer:** By *R6b* the set Orders is dynamic since new orders can be added and old orders can be deleted. Therefore, functions referencedProduct and orderQuantity are both dynamic and updated when a new order is inserted in Orders. The set Products is still assumed to be static since in *R6b* the entry of new products is not considered. The function stockQuantity is still dynamic and updated not only when an order or a set of orders is invoiced but also when new quantities of products are entered in the stock.

**Question 13:** How and by which *transitions* (actions) do system states evolve? How are the "internal" actions (of the system) related to "external" actions (of the environment)?

**Answer:** Besides the action of invoicing an order, *R6b* introduces other three operations: (1) cancelation of orders, (2) insertion of new orders, and (3) addition of quantities of products in the stock. We assume that these operations are driven by the monitored functions ordersToCancel, newOrders and newItems which return a sequence. The requested actions will be performed for every element in the sequence at each step. If the sequence is empty, the action has no effect. We introduce the following rule which is in charge of the cancelation of orders:

```
rule r_CancelOrders =
  forall $i in Natural with $i < length(ordersToCancel) do
      orderState(at(ordersToCancel,$i)) := CANCELED
```

Note that an order may be canceled even if it is already INVOICED. To allow only the cancelation of pending orders, the update of the order state must be guarded by orderState(at(ordersToCancel,$i))!= INVOICED.

### Extending domains

So far we have updated locations, i.e. changed the value of functions on existing elements. If we want to introduce new orders in the Orders set, then we have to create or construct new orders. To construct new elements and to add them to domains, ASMs introduces the extend notation:

**extend** $D$ **with** $v$ **do** $R(v)$

where *D* is the name of the abstract type-domain to be extended, *v* is the logical variable which is bound to the new element imported in *D* from the *reserve* (see [BÖR 03]) and *R* is a transition rule executed after *v* is added to *D*. Generally *R* will perform some initialisation over *v*.

In order to deal with the problem of incoming new orders, we need to answer the following question:

**Question 14:** Could the domains be extended by adding new items? Namely, in the case of invoicing orders, can new orders be inserted?

**Answer:** We answer the question by the following rule AddOrders which extends the domain Orders with new elements and sets all functions on these new locations:

```
rule r_AddOrders =
 forall $i in Natural with $i < length(newOrders) do
  let   $p = first(at(newOrders,$i)),
        $q = second(at(newOrders,$i)) in
   extend Orders with $order do par
         orderQuantity($order) :=$q
         referencedProduct($order) :=$p
         orderState($order) := PENDING
   endpar
```

**Sequentialisation and iteration**

The characteristics of basic ASMs (simultaneous execution of multiple atomic actions in a global state) come at a price, namely the lack of direct support for practical composition and structuring principles. *Turbo ASMs* offer as building blocks sequential composition, iteration and parametrised (possibly recursive) sub-machines extending the macro notation used with basic ASMs. They capture the sub-machine notions in a black-box view hiding the internals of sub-computations by compressing them into one step. A Turbo ASM can be obtained from basic ASMs by applying finitely often and in any order the operators of *sequential composition*, *iteration* and *sub-machine call*. We report here only the definition of the *seq* and *iterate* constructors which we need for our purposes (namely to deal with the problem of incoming new items; see below). A complete overview of the Turbo ASMs can be found in [BÖR 03].

We denote the *sequential composition* of two ASM rules *P* and *Q* by $P$ **seq** $Q$ and define its semantics as the effect of first executing *P* in a given state *S* and then *Q* in the resulting (invisible micro-)state $S + U$ (if it is defined), where *U* is the set of updates produced by *P* in *S*. *Q* may overwrite a location which has been updated by *P*. The set of updates produced by *P* and then *Q* are merged only if *U* is consistent, so obtaining the new state $S'$; otherwise $S'$ is the effect of applying *U* on *S*.

The construct **iterate** *R* iterates the sequential execution of a rule *R* encapsulating computations with a finite number of iterated steps into one step. It is defined by $R_0$ = **skip** (i.e. do nothing) and $R_{n+1} = R_n$ **seq** $R$. For iterated rule applications with *a priori* fixed bounds, we use the construct **while** (*cond*) *R* (= **iterate** (**if** *cond* **then** *R)*) when the stopping condition is specified, or **iterate** $v$ **in** $D$ **with** $G_v$ **do** $R(v)$ to express the subsequent execution of a rule $R$ for each $v$ satisfying a given condition $G$. There are two natural stop situations for iterated rule applications without *a priori* fixed bounds, namely when the update set becomes empty (the case of *successful termination*) and when it becomes inconsistent (the case of *failure*).

We exploit the last form of the construct iterate to deal with the problem of entering new items. Requirements do not guarantee that two (or more) entries of a same product cannot arrive at the same time, so inconsistent updates may arise. The question is:

**Question 15:** How can location updates be sequentialized in order to avoid synchronous inconsistent updating? In the case study, how can the stock be updated when new quantities for the same product arrive at the same time?

**Answer:** The following rule AddItems performs the entry of quantities in the stock by increasing the value of the function stockQuantity for the entered products. Since the monitored sequence newItems could contain the same product several times, the function stockQuantity cannot be updated in parallel for each product in the sequence, otherwise inconsistent updates may appear (unless one assumes that a same product occurs no more than ones in the list newItems):

```
rule r_AddItems =
    iterate $i in Natural with $i < length(newItems) do
      let $p = first(at(newItems,$i)), $q = second(at(newItems,$i)) in
          stockQuantity($p) := stockQuantity($p) + $q
```

The three new rules CancelOrders, AddOrders and AddItems respectively update the function orderState for existing orders, the domain Orders, and the function stockQuantity. Therefore, they can be executed in parallel. The fourth action of the system to invoice orders (described in case 1) updates the functions orderState and stockQuantity, hence it cannot be executed in parallel with rules CancelOrders and AddItems. Some form of synchronization or scheduling must be introduced. Since this information is missing in the requirements, we decide to execute the first three actions in parallel and then perform the rule that invoices orders. The following main rule orderSystem which formalises the whole system behavior, reports the rule InvoiceOrders. However, any other rule presented in section 6.2.4 can be replaced according with the chosen selection strategy discussed for case 1.

```
main rule r_orderSystem =
  seq
    par
      r_AddOrders()
      r_CancelOrders()
      r_AddItems()
    endpar
    r_InvoiceOrders()
  endseq
```

## 6.4 The natural language description of the specification

### 6.4.1 Case 1

The system of invoicing orders is a single-agent machine. There is a set Orders which is static, namely new orders cannot be added, and every order has a state, which can be *invoiced* or *pending*. All the orders are initially pending. There is a set of products and new products cannot be added. Every order refers to a product for a certain quantity (greater than zero) and these data cannot be changed. The same product can be referenced by several different orders. Every product is in the stock in different quantity. The quantity of a product in the stock is only updated by the system when it invoices some orders. The system selects a set of orders which are invoicable, i.e. they are pending and refer to a product in the stock in enough quantity, it simultaneously changes the state of each order in this set from pending to invoiced, and updates the stock by subtracting the total product quantity in orders to invoice. The system keeps to invoice orders as long as there are orders which can be invoiced. The system guarantees that the state of an order is always defined and the stock quantity is always greater than or equal to zero.

### 6.4.2 Case 2

For the new operations foreseen in this case of *canceling orders*, *entering new orders*, and *adding new quantities of products in the stock*, the system takes three inputs: ordersToCancel, a sequence of orders to cancel, newOrders, a sequence of orders to add (as a sequence of pairs product and quantity), and newItems, which gives the new quantities to add in the stock (as a sequence of pairs product and quantity).

At every computation step, all the orders in ordersToCancel are not really deleted, but their status changed to CANCELED. Since new orders can be entered, the set Orders is dynamic in this case and all the orders in newOrders set are inserted in Orders in one step. The reference to a product and the quantity for a new order are set when this new order is entered. Furthermore, the system updates the stock quantities for all the products in newItems in one step taking into account the total quantity when the same product is present several times in newItems. The three new operations are performed in parallel. The fourth action of invoicing orders (described in case 1) is executed afterwards.

## 6.5 Conclusion

Elicitation of requirements is a notoriously difficult and most error-prone part of the system development activities. Requirements capture is largely a formalisation task, namely to realize the transition from natural language problem descriptions – which are often incomplete or interspersed with misleading details, partly ambiguous or even inconsistent – to a sufficiently precise, unambiguous, consistent, complete and minimal description which can serve as a basis for the *contract* between the customer or

domain expert and the software designer. We have showed how the ASM method makes it possible to capture informal requirements by constructing a consistent and unambiguous, simple and concise, abstract and complete *ground model* which can be understood and checked (for correctness and completeness) by both domain experts and system designers.

During the formalisation process we have shown how requirements are often incomplete and assumptions must be stated in order to complete the specification. We have also shown how the ASM method is suitable to adapt the specification when different interpretations of the same requirements are possible (i.e. the discussion on different selection strategies of orders to be invoiced), and how the rigor of the ASM ground model allows formal (automatic) verification of properties. Furthermore, the documentation can be easily rephrased in natural language for an intuitive understanding of the formal description.

**Bibliography**

[ASMM] "The Abstract State Machines Metamodel (AsmM) website", http://www.dti.unimi.it/∼riccobene/asmm/.

[BÖR 03] BÖRGER E., STÄRK R., *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.

[BÖR 05] BÖRGER E., "The ASM Method for System Design and Analysis. A Tutorial Introduction", in GRAMLICH B., Ed., *FroCoS 2005*, vol. 3717 of *Lecture Notes in Artificial Intelligence*, Vienna (Austria), Springer, p. 264-283, September 2005.

[DEL 00] DEL CASTILLO G., WINTER K., "Model Checking Support for the ASM High-Level Language", in GRAF S., SCHWARTZBACH M., Eds., *Proc. of TACAS*, vol. 1785 of *LNCS*, Springer-Verlag, p. 331–346, 2000.

[GAR 00] GARGANTINI A., RICCOBENE E., "Encoding Abstract State Machines in PVS", in GUREVICH Y., KUTTER P., ODERSKY M., THIELE L., Eds., *Abstract State Machines – Theory and Applications: International Workshop, ASM 2000*, vol. 1912 of *LNCS*, Monte Verità, Switzerland, Springer, p. 303–322, March 2000.

[SCA 05] SCANDURRA P., GARGANTINI A., GENOVESE C., GENOVESE T., RICCOBENE E., "A concrete syntax derived from the Abstract State Machine metamodel", in *Proc. of Abstract State Machines 2005*, 2005.